

Decision Procedure for A Theory of String Sequences

Denghang Hu^{1,2}, Taolue Chen³, Philipp Rümmer⁴,
Fu Song^{1,2,5}, and Zhilin Wu^{1,2}

¹ Key Laboratory of System Software and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China

² University of Chinese Academy of Sciences, China

³ Birkbeck, University of London, United Kingdom

⁴ University of Regensburg, Germany

⁵ Nanjing Institute of Software Technology, China

Abstract. The theory of sequences, supported by many SMT solvers, can model program data types including bounded arrays and lists. Sequences are parameterized by the element data type and provide operations such as accessing elements, concatenation, forming sub-sequences and updating elements. Strings and sequences are intimately related; many operations, e.g., matching a string according to a regular expression, splitting strings, or joining strings in a sequence, are frequently used in string-manipulating programs. Nevertheless, these operations are typically not directly supported by existing SMT solvers, which instead only consider the generic theory of sequences. In this paper, we propose a theory of string sequences and study its satisfiability. We show that, while it is undecidable in general, the decidability can be recovered by restricting to the straight-line fragment. This is shown by encoding each string sequence as a string, and each string sequence operation as a corresponding string operation. We provide pre-image computation for the resulting string operations with respect to automata, effectively casting it into the generic OSTRICH string constraint solving framework. We implement the new decision procedure as a tool `OSTRICHSEQ`, and carry out experiments on benchmark constraints generated from real-world JavaScript programs, hand-crafted templates and unit tests. The experiments confirm the efficacy of our approach.

1 Introduction

Many real-world applications, such as web applications and programs processing data of type string, involve a multitude of complex operations that convert between strings and string sequences. Typical examples include `join` and `split`, which are present in almost all built-in libraries of modern programming languages. Reasoning about string sequences, also known as extendable string arrays [33], is therefore a crucial aspect of program analysis.

Satisfiability Modulo Theory (SMT [9]) solving provides an automatic way to verify software systems. At the moment, there is no standardized sequence theory

in SMT-LIB [8]. Instead, in verification tools, often the theory of arrays is used, which is limited and does not provide even basic operations like sequence length. Fortunately, [14] proposed a basic format for a sequence theory, which has been implemented and extended by SMT solvers such as Z3 [25] and cvc5 [7] with practical operations. In particular, [44] presents a calculus in cvc5 for reasoning about string sequences, which extends the proposal [14] with `update` (aka `write`) operations. The implementation of sequences in Z3 has been documented in the Z3 guide [50] and includes operations such as `map` and `foldleft` (but no `update`).

Although existing SMT solvers are effective at reasoning about sequences in general, to the best of our knowledge, none of them directly support functions that convert between strings and string sequences such as `join` and `split`. While it is possible to implement these functions using existing ones for some SMT solvers (e.g., Z3 can use `map` and `foldleft` to implement the `join` function), usually the solvers do not provide decision procedures for those theories and fail to solve many instances. This limitation poses a significant challenge for program analysis that involves such operations. The purpose of our work is to fill this gap.

Contributions. We propose a logic (aka constraint language) `SeqStr` of string sequences, which extends the existing sequence theory in SMT solvers with dedicated operations of string sequences and strings. In addition to the standard string operations (`replace/replaceAll`, `reverse`, finite transducers, regular constraints, `length`, `indexOf`, `substring`, etc.), our logic emphasizes the interaction between strings and string sequences through operations such as `split`, `join`, `filter`, and `matchAll`. For instance, `split` splits a string into a sequence of substrings according to a regular expression; `join` concatenates all the strings in a sequence to obtain a single string. More details can be found in Section 4. Typically, when string sequences are considered, it is natural to consider the integer data type as well; for instance, one usually needs to refer to the length of a sequence or the index of a specific sequence element. As a result, `SeqStr` features three data types: integers, strings, and string sequences. In terms of operations, the logic subsumes, to our best knowledge, most string constraint languages that have been proposed in the literature.

Our main focus is the decision procedure for the satisfiability problem of `SeqStr`. Because of its expressiveness, it is perhaps not surprising that `SeqStr` is undecidable in general. To reinstate decidability, we consider the straight-line fragment, which imposes a syntactic restriction on the constraints written in the logic. Straight-line formulas naturally arise when verifying programs using bounded model checking or symbolic execution, which unroll loops in the programs up to a given depth and convert programs to static single assignment form (i.e. each variable is defined at most once). For example, a majority of the constraints in the standard Kaluza benchmarks [42] satisfy this condition. (Cf. Section 3 for a concrete example.)

The general strategy of our decision procedure is to encode each string sequence as a string in such a way that all operations for string sequences considered in `SeqStr` can be transformed into string operations, possibly involving integers. As such, the decidability of (straight-line) `SeqStr` is reduced to string

constraints with the integer data type, which was shown to be decidable [19]. Needless to say, this strategy requires overcoming certain technical challenges, as the string constraints resulting from the reduction typically encompass complex (and sometimes non-standard) string operations, which go well beyond the capacity of state-of-the-art string constraint solvers. Recall that for the decidability of string constraints with integers, cost-enriched finite automata (CEFA), a variant of cost-register automata, were utilized [19]. The crux of our technical contributions is thus to compute the backward images of CEFAs under the fairly complex string operations (cf. Section 5.2), so the powerful string constraint solving framework OSTRICH [17,20] can be harnessed.

We implement the decision procedure as a new solver $\text{OSTRICH}^{\text{SEQ}}$ on top of OSTRICH [20] and Princess [40]. To evaluate the effectiveness of $\text{OSTRICH}^{\text{SEQ}}$, we curate two benchmark suites of constraints that are randomly generated from templates (only using the operations directly supported by some existing SMT solvers) and extracted from real-world JavaScript programs and unit tests (involving string sequence operations that are not directly supported by existing SMT solvers), respectively. On both benchmark suites, the experimental results show that $\text{OSTRICH}^{\text{SEQ}}$ can solve considerably more constraints than SOTA string solvers including *cvc5*, *Z3*, *Z3-noodler* [22], OSTRICH and Princess^{ARR} [41], while being largely as efficient as most of them.

Organization. Section 3 presents a motivating example. Section 2 gives the preliminaries. Section 4 defines the logic *SeqStr* of string sequences. Section 5 presents the decision procedure. Section 6 presents the benchmarks and experiments. Section 7 discusses the related work. The paper is concluded in Section 8.

2 Preliminaries

Let \mathbb{N} denote the set of natural numbers. For $1 \leq n \in \mathbb{N}$, let $[n] := \{1, \dots, n\}$, and for $m < n \in \mathbb{N}$, let $[m, n] := \{j \mid m \leq j \leq n\}$. We also use standard quantifier-free/existential *linear integer arithmetic* (LIA) formulas, which are typically ranged over by ϕ, φ , etc. For an integer term t , we use $t[t_1/t_2]$ to denote the term obtained by replacing t_2 with t_1 where t_1 and t_2 are integer terms.

Strings, languages, and transductions. We fix an alphabet Σ , i.e., a finite set of letters. A *string* over Σ is a finite sequence of letters from Σ . We use Σ^* to denote the set of strings over Σ and ε to denote the empty string. A *language* over Σ is a subset of Σ^* . We will use L_1, L_2, \dots to denote languages. For two languages $L_1, L_2 \subseteq \Sigma^*$, $L_1 \cup L_2$ denotes the union of L_1 and L_2 , and $L_1 \cdot L_2$ denotes the concatenation of L_1 and L_2 , that is, $\{u_1 \cdot u_2 \mid u_1 \in L_1, u_2 \in L_2\}$. For a language $L \subseteq \Sigma^*$, we define the complement of L as $\bar{L} = \{w \in \Sigma^* \mid w \notin L\}$, moreover, we define L^n for $n \in \mathbb{N}$, the *iteration* of L for n times, inductively as: $L^0 = \{\varepsilon\}$ and $L^n = L \cdot L^{n-1}$ for $n > 0$. We also use L^* to denote the iteration of L for arbitrarily many times, that is, $L^* = \bigcup_{n \in \mathbb{N}} L^n$, and let $L^+ = \bigcup_{n > 0} L^n$. A *transduction* over Σ is a binary relation over Σ^* , namely, a subset of $\Sigma^* \times \Sigma^*$.

Regular expressions over Σ are defined in a standard way, i.e.,

$$e \stackrel{\text{def}}{=} \emptyset \mid \varepsilon \mid a \mid e + e \mid e \cdot e \mid e^* \text{ where } a \in \Sigma.$$

The language $\mathcal{L}(e) \subseteq \Sigma^*$ of a regular expression e is defined inductively as: $\mathcal{L}(\emptyset) = \emptyset$, $\mathcal{L}(\varepsilon) = \{\varepsilon\}$, $\mathcal{L}(a) = \{a\}$, $\mathcal{L}(e_1 + e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$, $\mathcal{L}(e_1 \cdot e_2) = \mathcal{L}(e_1) \cdot \mathcal{L}(e_2)$, $\mathcal{L}(e_1^*) = (\mathcal{L}(e_1))^*$. A *regular language* is a language that can be defined by a regular expression. Moreover, \bar{e} is the complement of e , that is, $\mathcal{L}(\bar{e}) = \{w \mid w \notin \mathcal{L}(e)\}$.

Automata. A (*nondeterministic*) *finite automaton* (NFA) \mathcal{A} is a 5-tuple $(Q, \Sigma, \delta, I, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $I, F \subseteq Q$ are the sets of initial and final states respectively. For readability, we write a transition $(q, a, q') \in \delta$ as $q \xrightarrow{a} q'$ (or simply $q \xrightarrow{a} q'$). A *run* of \mathcal{A} on a string $w = a_1 \cdots a_n$ is a sequence of transitions $q_0 \xrightarrow{a_1} q_1 \cdots q_{n-1} \xrightarrow{a_n} q_n$ with $q_0 \in I$. The run is *accepting* if $q_n \in F$. A string w is accepted by an NFA \mathcal{A} if there is an accepting run of \mathcal{A} on w . In particular, the empty string ε is accepted by \mathcal{A} if $I \cap F \neq \emptyset$. The language of \mathcal{A} , denoted by $\mathcal{L}(\mathcal{A})$, is the set of strings accepted by \mathcal{A} . In addition, an NFA can be built to recognize the language of each given regular expression, and vice versa.

A (*nondeterministic*) *finite transducer* (NFT) \mathfrak{T} is an extension of NFA with outputs. Formally, an NFT \mathfrak{T} is a 5-tuple $(Q, \Sigma, \delta, I, F)$, where Q, Σ, I, F are the same as in NFA and the transition relation δ is a finite subset of $Q \times \Sigma \times Q \times \Sigma^*$. For readability, we write a transition $(q, a, q', u) \in \delta$ as $q \xrightarrow[a]{a, u} q'$ or $q \xrightarrow{a, u} q'$.

A run of \mathfrak{T} over a string $w = a_1 \cdots a_n$ is a sequence of transitions $q_0 \xrightarrow{a_1, u_1} q_1 \xrightarrow{a_2, u_2} q_2 \cdots q_{n-1} \xrightarrow{a_n, u_n} q_n$ where $q_0 \in I$. Similar to NFA, the run is *accepting* if $q_n \in F$. The string $u_1 \cdots u_n$ is called the *output* of the run. The transduction $\mathcal{T}(\mathfrak{T}) \subseteq \Sigma^* \times \Sigma^*$ defined by \mathfrak{T} is the set of string pairs (w, u) such that there is an accepting run of \mathfrak{T} on w , with the output u .

Definition 1 (Cost-enriched finite automata). A *cost-enriched finite automaton* (CEFA for short) \mathcal{A} is a 6-tuple $(R, Q, \Sigma, \delta, I, F)$ where

- $R = \{r_1, \dots, r_k\}$ is a finite set of registers,
- Q, I, F are the same as in NFA, i.e., the set of states, the set of initial states, and the set of final states, respectively,
- $\delta \subseteq Q \times \Sigma \times Q \times \mathbb{Z}^k$ is a transition relation, where \mathbb{Z}^k represents the values to update the registers in R .

We write $R_{\mathcal{A}}$ for the set of registers of \mathcal{A} and represent it as a vector (r_1, \dots, r_k) . Accordingly, updates $r_i := r_i + v_i$ for all $i \in [k]$ are simply identified as the vector $\vec{v} = (v_1, \dots, v_k)$, i.e., r_i is incremented by v_i for each $i \in [k]$. Typically, we write a transition $(q, a, q', \vec{v}) \in \delta$ as $q \xrightarrow[\vec{v}]{a} q'$.

Intuitively, CEFAs add write-only cost registers to finite automata, where “write-only” means that the cost registers can only be written/updated but cannot be read, i.e., they cannot be used in the guards of the transitions.

A *run* of the CEFA \mathcal{A} on a string $w = a_1 \cdots a_n$ is a sequence of transitions $q_0 \xrightarrow[\vec{v}_1]{a_1} q_1 \cdots q_{n-1} \xrightarrow[\vec{v}_n]{a_n} q_n$ such that $q_0 \in I$ and $q_{i-1} \xrightarrow[\vec{v}_i]{a_i} q_i$ for each $i \in [n]$. It is *accepting* if $q_n \in F$, and the vector $\vec{c} = \sum_{i \in [n]} \vec{v}_i$ is defined as the *cost* of the run. (Note that all registers are initialized to zero.) We write $\vec{c} \in \mathcal{A}(w)$ if there

is an accepting run of \mathcal{A} on w whose cost is \vec{c} . The language of a CEFA \mathcal{A} , denoted by $\mathcal{L}(\mathcal{A})$, is defined as the set of pairs $\{(w, \vec{c}) \in \Sigma^* \times \mathbb{Z}^{|R|} \mid \vec{c} \in \mathcal{A}(w)\}$. In particular, if $I \cap F \neq \emptyset$, then $(\varepsilon, \vec{0}) \in \mathcal{L}(\mathcal{A})$. We denote by $\mathcal{L}_1(\mathcal{A})$ the language $\{w \in \Sigma^* \mid \exists \vec{c} \in \mathbb{Z}^{|R|}. (w, \vec{c}) \in \mathcal{L}(\mathcal{A})\}$.

Pre-images under string functions. Consider a language $L \subseteq \Sigma^* \times \mathbb{Z}^{k_0}$ defined by a CEFA $\mathcal{A} = (R, Q, \Sigma, \delta, I, F)$ with the registers $R = (r_1, \dots, r_{k_0})$ and a function $f : (\Sigma^* \times \mathbb{Z}^{k_1}) \times \dots \times (\Sigma^* \times \mathbb{Z}^{k_l}) \rightarrow \Sigma^*$. For each $i \in [k_0]$, let r_1^i, \dots, r_l^i be the freshly introduced registers, and $\vec{t} = (t_1, \dots, t_{k_0})$ be a vector of LIA formulas such that t_i is a linear combination of r_1^i, \dots, r_l^i for each $i \in [k_0]$.

The pre-image of L under f with respect to \vec{t} , denoted by $f_{\vec{t}}^{-1}(L)$, is a relation

$$\mathcal{R} \subseteq (\Sigma^* \times \mathbb{Z}^{k_1+k_0}) \times \dots \times (\Sigma^* \times \mathbb{Z}^{k_l+k_0})$$

that comprises tuples of the form $((w_1, (\vec{c}_1, \vec{d}_1)), \dots, (w_l, (\vec{c}_l, \vec{d}_l)))$ such that

- for every $j \in [l]$, $\vec{c}_j \in \mathbb{Z}^{k_j}$ and $\vec{d}_j = (d_j^1, \dots, d_j^{k_0}) \in \mathbb{Z}^{k_0}$,
- let $w_0 = f((w_1, (\vec{c}_1, \vec{d}_1)), \dots, (w_l, (\vec{c}_l, \vec{d}_l)))$ and

$$\vec{d}' = \left(t_1 [d_1^1/r_1^1, \dots, d_l^1/r_l^1], \dots, t_{k_0} [d_1^{k_0}/r_1^{k_0}, \dots, d_l^{k_0}/r_l^{k_0}] \right),$$

it holds that $(w_0, \vec{d}') \in L$.

The pre-image $f_{\vec{t}}^{-1}(L)$ is *CEFA-definable* if $f_{\vec{t}}^{-1}(L) = \bigcup_{i=1}^n \mathcal{L}(\mathcal{A}_{i,1}) \times \dots \times \mathcal{L}(\mathcal{A}_{i,l})$ for some $n \geq 1$, where for all $i \in [n]$ and $j \in [l]$, $\mathcal{A}_{i,j}$ is a CEFA such that $\mathcal{L}(\mathcal{A}_{i,j}) \subseteq \Sigma^* \times \mathbb{Z}^{k_j}$. Finally, a pre-image of L under f , denoted by $f^{-1}(L)$, is a pair (\mathcal{R}, \vec{t}) such that $\mathcal{R} = f_{\vec{t}}^{-1}(L)$.

The core concept of *pre-image computation* for the function f involves determining the vector \vec{t} of LIA formulas and, for each CEFA \mathcal{A} , computing $f_{\vec{t}}^{-1}(\mathcal{L}(\mathcal{A}))$ represented as a finite set of CEFAs. Intuitively, we remove the string equalities of the form $y = f(x_1, it_1, \dots, x_l, it_l)$ (where it_1, \dots, it_l are integer expressions) one by one by computing the pre-images and adding the resulting CEFA membership constraints for x_1, \dots, x_l . In the end, the original string constraint is transformed into a conjunction of CEFA membership constraints and LIA formulas, whose satisfiability checking is known to be PSPACE-complete [19].

3 Motivating Example

To motivate and illustrate our approach, we consider a JavaScript snippet shown in Listing 1.1, which is extracted, with a slight adaptation, from a real-world example.⁶ Here, the function `prepareVersionNo` extracts from the input string `version` a version number in the floating-point number format. Specifically, it splits `version` into a sequence/array `numbers` according to the non-numeric characters. If `numbers` contains exactly one element, `numbers[0]` is returned; otherwise, the function returns the concatenation of `numbers[0]`, the dot, and

⁶ <https://github.com/hgoebl/mobile-detect.js/blob/master/mobile-detect.js>

```

1  function prepareVersionNo(version) {
2      // precondition: version in [0-9]+([0-9a-zA-Z._ /-])*
3      numbers = version.split(/[a-zA-Z._ /-]/i);
4      if (numbers.length === 1) { // path-1
5          result = numbers[0];      }
6      if (numbers.length > 1) { // path-2
7          temp = numbers[0] + ".";
8          numbers1 = numbers.slice(1, numbers.length);
9          result = temp + numbers1.join("");      }
10     return Number(result);
11     // postcondition: result in [0-9]+(\.[0-9]*)?
12 }

```

Listing 1.1. JavaScript code snippet: The motivating example

```

1  ; precondition
2  (assert (str.in_re version preReg))
3  (assert (= numbers (str.splitre version splitReg)))
4  (assert (< 1 (seq.len numbers)))
5  (assert (= temp (str.++ (seq.nth numbers 0) ".")))
6  (assert (= numbers1
7          (seq.extract numbers 1 (- (seq.len numbers) 1))))
8  (assert (= result (str.++ temp (seq.join numbers1 ""))))
9  ; postcondition
10 (assert (not (str.in_re result postReg)))

```

Listing 1.2. SMT formula for path-2 in prepareVersionNo

the string obtained by concatenating the other elements of `numbers`. For instance, if `version = 12a56b23`, the output is 12.5623.

To verify the functional correctness of `prepareVersionNo`, we specify the following precondition and postcondition.

- The precondition `version ∈ [0-9]+([0-9a-zA-Z._ /-])*` requires that the input string starts with a decimal number, followed by a string of digits, letters, dot `.`, underline `_`, blank symbol, slash `/`, or hyphen `-`.
- The postcondition `result ∈ [0-9]+(\.[0-9]*)?` ensures that the output string starts with a decimal number, possibly followed by a dot `.` and a decimal number.

We apply symbolic execution to verify `prepareVersionNo`. We enumerate its execution paths and check for each path that, provided the input string satisfies the precondition, the output string satisfies the postcondition after the executing the path. There are three execution paths: path-1 taking `numbers.length===1`, path-2 taking `numbers.length>1`, and path-3 where both conditions are false. As an example, we consider path-2, and the symbolic execution reduces to deciding the satisfiability of the SMT formula in Listing 1.2, where `preReg` and `postReg` denote the regular expressions in the precondition and postcondition, i.e., `([0-9]+([0-9a-zA-Z._ /-])*)` and `([0-9]+(\.[0-9]*)?)`, respectively, and `splitReg` represents the regular expression `[a-zA-Z._ /-]`.

This SMT formula involves various operations of string sequences and strings, including sequence length `seq.len`, string split `str.splitre`, sequence read `seq.nth`, string concatenation `str.++`, sequence extract `seq.extract`, sequence join `seq.join`, as well as regular constraints `str.in_re`. While state-of-the-art SMT solvers such as Z3 and cvc5 can solve constraints in the generic sequence theory, they do not directly support complex operations that feature mutual transformations between strings and string sequences, such as `str.splitre` and `seq.join`. This motivates us to define an SMT theory of string sequences (SeqStr, cf. Section 4) and investigate its decision procedure (Section 5).

Due to the undecidability of SeqStr, we focus on its straight-line fragment (cf. Definition 2), into which the formula in Listing 1.2 falls. We encode string sequences as strings, and, accordingly, string sequence operations become string operations (Section 5.1). Then we transform the formula in Listing 1.2 into the following string constraint:

$$\begin{aligned}
& \text{version} \in \text{preReg} \wedge \text{snumbers} = \text{splitstr}_{\text{splitReg}}(\text{version}) \wedge \\
& 1 < \text{seqlen}(\text{snumbers}) - 1 \wedge \text{temp} = \text{elem}(\text{snumbers}, 0) \cdot '.' \wedge \\
& \text{snumbers1} = \text{subseq}[\text{snumbers}, 2, \text{seqlen}(\text{snumbers}) - 2] \wedge \\
& \text{result} = \text{temp} \cdot \text{join}_\epsilon(\text{snumbers1}) \wedge \text{result} \notin \text{posReg}.
\end{aligned} \tag{1}$$

where `snumbers`, `snumbers1` are string variables that encode the string sequences `numbers`, `numbers1` in Listing 1.2. Note that the resulting string constraint contains new string operations `splitstr`, `seqlen`, `elem`, \dots introduced in this paper (cf. Section 5.1). Subsequently, we employ the decision procedure outlined in [19], which *back propagates* the CEFA membership constraints by computing the pre-images under string functions (see Section 5.2).

In the sequel, we show how to solve the formula in (1).

1. We back propagate the regular membership constraint $\text{result} \in \overline{\text{posReg}}$ (where $\overline{\text{posReg}}$ denotes the complement of `posReg`) by computing the pre-images under the concatenation \cdot and join_ϵ , and adding the CEFA constraints for `temp` and `snumbers1`, say, $\text{temp} \in \mathcal{A}_1 \wedge \text{snumbers1} \in \mathcal{A}_2$ (where $\text{temp} \in \mathcal{A}_1$ means that the value of `temp` belongs to $\mathcal{L}_1(\mathcal{A}_1)$, similarly for $\text{snumbers1} \in \mathcal{A}_2$). Note that nondeterministic choices may be made here since the pre-images of $\mathcal{L}(\overline{\text{posReg}})$ under \cdot may be a union of products of CEFAs. After the propagation, the string equality $\text{result} = \text{temp} \cdot \text{join}_\epsilon(\text{snumbers1})$ is removed.
2. Then we back propagate the CEFA membership constraint $\text{temp} \in \mathcal{A}_1$ for the equality $\text{temp} = \text{elem}(\text{snumbers}, 0) \cdot '.'$ by computing the pre-image of $\mathcal{L}(\mathcal{A}_1)$ under \cdot as well as `elem` and adding some CEFA membership constraint for `snumbers`, say, $\text{snumbers} \in \mathcal{A}_3$. Similarly, we back propagate the CEFA membership constraint $\text{snumbers1} \in \mathcal{L}(\mathcal{A}_2)$ for the equality $\text{snumbers1} = \text{subseq}[\text{snumbers}, 2, \text{seqlen}(\text{snumbers}) - 1]$ by computing the pre-image of $\mathcal{L}(\mathcal{A}_2)$ under `subseq` and adding some CEFA membership constraint for `snumbers`, say, $\text{snumbers} \in \mathcal{A}_4$. After these back-propagation operations, the two string equalities are removed.

3. Finally, we back propagate the CEFA membership constraint $\mathbf{snumbers} \in \mathcal{A}_3 \cap \mathcal{A}_4$ for $\mathbf{snumbers} = \text{splitstr}_{\text{splitReg}}(\mathbf{version})$ by computing the pre-image of $\mathcal{L}(\mathcal{A}_3 \cap \mathcal{A}_4)$ under the function $\text{splitstr}_{\text{splitReg}}$ and adding a CEFA membership constraint $\mathbf{version} \in \mathcal{A}_5$. Moreover, the equality $\mathbf{snumbers} = \text{splitstr}_{\text{splitReg}}(\mathbf{version})$ is removed.
4. In the end, we obtain a string constraint that contains no string equalities, namely, a Boolean combination of CEFA membership constraints and LIA formulas, which are solvable using existing methods.

Note that here we prioritize the illustration of the main idea over the precision of the description; the technical details of the decision procedure are deferred to later sections.

Overall, the formula in Listing 1.2 can be transformed into a string constraint containing a complex combination of `str.replace_re_all`, `str.substr` and `str.len`, which cannot be solved by Z3 or cvc5 in a reasonable time limit (Z3 returns unknown as it does not support the `str.replace_re_all` function; cvc5 fails to solve the string constraint in 20 hours). In contrast, our new solver OSTRICH^{SEQ} solves it in less than 1 second.

4 A Theory of String Sequences (SeqStr)

A *sequence* over the set X is (a_0, \dots, a_{n-1}) , where $a_i \in X$ for each $i \in [0, n-1]$, and n is the *length* of the sequence. We shall focus on *string sequences*, namely, sequences over Σ^* . We use $()$ to denote the empty sequence.

Operations on string sequences and strings. We assume two string sequences $s_1 = (u_0, \dots, u_{m-1})$ and $s_2 = (v_0, \dots, v_{n-1})$.

- $s_1 \cdot s_2$ is the concatenation of s_1 and s_2 , i.e., $(u_0, \dots, u_{m-1}, v_0, \dots, v_{n-1})$.
- $\text{nth}(s_1, i)$ is u_i if $i \in [0, m-1]$, and undefined otherwise.
- $\text{join}_v(s_1)$ joins the elements in s_1 with v as the intermediate string, i.e., $\text{join}_v(s_1) = u_0 v u_1 v u_2 v \dots v u_{m-1}$.
- $s_1[i \rightarrow u]$ replaces u_i in s_1 by u if $i \in [0, m-1]$, and is s_1 otherwise, i.e.

$$s_1[i \rightarrow u] = \begin{cases} (u_0, \dots, u_{i-1}, u, u_{i+1}, \dots, u_{m-1}), & \text{if } i \in [0, m-1]; \\ s_1, & \text{otherwise.} \end{cases}$$

- $\text{filter}_e(s_1)$ filters out all elements in s_1 that do not match the regular expression e , i.e., $\text{filter}_e(s_1) = (u_{i_1}, \dots, u_{i_k})$ such that $0 \leq i_1 < \dots < i_k \leq m-1$ and for all $j \in [0, m-1]$, $u_j \in \mathcal{L}(e)$ iff $j \in \{i_1, \dots, i_k\}$. In particular, if none of u_0, \dots, u_{m-1} are in $\mathcal{L}(e)$, then $\text{filter}_e(s_1) = ()$.
- $\text{split}_e(u)$ splits a string u into a sequence of substrings according to the regular expression e , i.e., $\text{split}_e(u) = (u'_1, \dots, u'_k)$, where $u = u'_1 v_1 u'_2 v_2 \dots u'_{k-1} v_{k-1} u'_k$ such that (1) for each $i \in [k-1]$, v_i is the leftmost and longest substring of $u'_i v_i u'_{i+1} \dots v_{k-1} u'_k$ that matches e , and (2) u'_k does not contain any substring that matches e . In particular, if u does not contain any substring that matches e , then $\text{split}_e(u) = (u)$.

- $s_1[i, j]$ is defined as follows: if $i \in [0, m-1]$ and $j \geq 1$, then $s_1[i, j]$ is the subsequence $(u_i, \dots, u_{\min(i+j-1, m-1)})$; if $i \in [0, m-1]$ and $j = 0$, then $s_1[i, j]$ is the empty sequence $()$; if $i \notin [0, m-1]$ or $j < 0$, then $s_1[i, j]$ is undefined.
- $\text{matchAll}_e(u)$ extracts the substrings of u that match the regular expression e , i.e., $\text{matchAll}_e(u) = (v_1, \dots, v_{k-1})$ such that $u = u'_1 v_1 u'_2 v_2 \dots u'_{k-1} v_{k-1} u'_k$ and (1) for each $i \in [k-1]$, v_i is the leftmost and longest substring of $u'_i v_i u'_{i+1} \dots v_{k-1} u'_k$, and (2) u'_k has no substring that matches e .

Note that we choose the longest match semantics in $\text{split}_e(u)$ and $\text{matchAll}_e(u)$ just for illustrating our approach, where the shortest match semantics can be captured by adapting the automata construction in the pre-image computation.

The logic SeqStr of string sequences. SeqStr has three data types: Int (integer), Str (strings), and Seq (sequences). We use u, v, \dots (resp. $\mathbf{u}, \mathbf{v}, \dots$) to denote string constants (resp. variables); m, n, \dots (resp. $\mathbf{m}, \mathbf{n}, \dots$) to denote integer constants (resp. variables); s, t, \dots (resp. $\mathbf{s}, \mathbf{t}, \dots$) to denote sequence constants (resp. variables); and e to denote regular expressions.

The syntax of SeqStr is defined as follows:

$$\begin{array}{ll}
it \stackrel{\text{def}}{=} m \mid \mathbf{m} \mid it + it \mid it - it \mid \text{strlen}(strt) \mid \text{seqlen}(seqt) & \text{integer terms} \\
strt \stackrel{\text{def}}{=} u \mid \mathbf{u} \mid strt \cdot strt \mid \text{nth}(seqt, it) \mid \text{join}_u(seqt) & \text{string terms} \\
seqt \stackrel{\text{def}}{=} (strt) \mid s \mid \mathbf{s} \mid seqt \cdot seqt \mid seqt[it \rightarrow strt] \mid \text{filter}_e(seqt) \mid \\
\quad seqt[it, it] \mid \text{split}_e(strt) \mid \text{matchAll}_e(strt) & \text{sequence terms} \\
\varphi \stackrel{\text{def}}{=} it \bowtie it \mid seqt = seqt \mid strt = strt \mid strt \in e \mid \varphi \wedge \varphi & \text{formulas}
\end{array}$$

where $\bowtie \in \{=, \neq, <, \leq, >, \geq\}$.

Here, it , $seqt$ and $strt$ denote integer, sequence and string terms, respectively. In particular, the integer terms are linear arithmetic expressions where $\text{seqlen}(\mathbf{s})$ (resp. $\text{strlen}(\mathbf{u})$) denotes the length of a sequence \mathbf{s} (resp. string \mathbf{u}); the sequence terms are constructed from a singleton sequence, sequence constants and sequence variables by applying the aforementioned sequence/string operations. A SeqStr formula is a conjunction of atomic formulas, each of which is of the form $it_1 \bowtie it_2$ where $\bowtie \in \{=, \neq, <, \leq, >, \geq\}$, a sequence equality $seqt_1 = seqt_2$, or a string equality $strt_1 = strt_2$. Note that sequence and string inequalities can be expressed as well. For instance (using disjunctions and quantifiers for the sake of presentation):

$$\begin{aligned}
\mathbf{r} \neq \mathbf{\eta} &\equiv \exists \mathbf{r}_1, \mathbf{r}_2, \mathbf{\eta}_2. \bigvee_{a, b \in \Sigma, a \neq b} (\mathbf{r} = \mathbf{r}_1 a \mathbf{r}_2 \wedge \mathbf{\eta} = \mathbf{r}_1 b \mathbf{\eta}_2) \vee \exists \mathbf{z}. \bigvee_{a \in \Sigma} (\mathbf{r} = \mathbf{\eta} a \mathbf{z} \vee \mathbf{\eta} = \mathbf{r} a \mathbf{z}) \\
\alpha \neq \beta &\equiv (\exists \alpha_1, \alpha_2, \beta_2, \mathbf{z}_1, \mathbf{z}_2. \alpha = \alpha_1 \cdot (\mathbf{z}_1) \cdot \alpha_2 \wedge \beta = \alpha_1 \cdot (\mathbf{z}_2) \cdot \beta_2 \wedge \mathbf{z}_1 \neq \mathbf{z}_2) \vee \\
&\quad \exists \gamma, \mathbf{z}. \alpha = \beta \cdot (\mathbf{z}) \cdot \gamma \vee \beta = \alpha \cdot (\mathbf{z}) \cdot \gamma.
\end{aligned}$$

Though rewriting inequalities produces disjunctions, they can still be handled in the DPLL(T) framework [29]. Hence, we focus on the disjunction-free fragment.

Proposition 1. *The satisfiability of SeqStr is undecidable.*

We define the *straight-line fragment* $\text{SeqStr}_{\text{SL}}$ of SeqStr . It is easy to see that by introducing fresh variables, we can rewrite a SeqStr formula to a form in which the left-hand side of each equality $\text{seq}t_1 = \text{seq}t_2$ (resp. $\text{str}t_1 = \text{str}t_2$) is a sequence (resp. string) variable. Note that the original and rewritten formulas are equisatisfiable. Henceforth, we assume that all sequence/string equalities satisfy this constraint unless otherwise stated.

Definition 2 (Straight-line fragment). *Given a SeqStr formula φ , the dependency graph G_φ of φ is a directed graph (V, E) such that V is the set of string and sequence variables in φ , and $(v_1, v_2) \in E$ iff $v_1 = \mathbf{rhs}$ is a sequence or string equality in φ and v_2 occurs in \mathbf{lhs} except for integer terms.*

φ is straight-line if φ is disjunction-free, and each string/sequence variable occurs as the left-hand side of equalities at most once; moreover, G_φ is acyclic.

Example 1. $\mathbf{s} = \text{split}_e(\mathbf{u}) \wedge \mathbf{t} = \mathbf{s}[1, \text{seqlen}(\mathbf{s}) - 1]$ is straight-line, while $\mathbf{s} = \mathbf{s}[1, \text{seqlen}(\mathbf{s}) - 1]$ is not, since there is a self-dependency on \mathbf{s} .

5 The Decision Procedure

Theorem 1. *The satisfiability of $\text{SeqStr}_{\text{SL}}$ is decidable.*

The general idea of the decision procedure is to encode each string sequence as a string, based on which all string sequence operations in $\text{SeqStr}_{\text{SL}}$ can be transformed into string operations. It utilizes the framework [19] for solving straight-line string constraints with integer data type, as illustrated in Section 3. There are, however, certain technical challenges. For instance, $\text{seq}[it \rightarrow str]$ and $\text{seq}[it, it]$ cannot be captured directly by standard string operations, so new ones have to be provided whose pre-images need to be computed.

5.1 From string sequences to strings

We fix a symbol $\dagger \notin \Sigma$ and let $\Sigma_\dagger := \Sigma \cup \{\dagger\}$. Each string sequence $s = (u_1, \dots, u_m)$ over Σ is encoded as a string $\text{enc}(s) := \dagger u_1 \dagger u_2 \dagger \dots \dagger u_m \dagger$ over Σ_\dagger . Note that the strings resulted from the encoding should match the regular expression $\dagger(\Sigma^* \dagger)^*$ and the string \dagger encodes the empty sequence.

With this encoding of string sequences as strings, each string sequence operation is also transformed into the corresponding *string operation*.

sequence concatenation $s_1 \cdot s_2$: string concatenation $\text{enc}(s_1) \cdot \text{enc}(s_2)$.

sequence write $s[i \rightarrow u]$: $\text{write}(\text{enc}(s), i + 1, u)$, which replaces the substring of $\text{enc}(s)$ between the $(i + 1)$ -th occurrence of \dagger and the $(i + 2)$ -th occurrence of \dagger , by u , if $i \in [0, n - 2]$, where n is the number of occurrences of \dagger in $\text{enc}(s)$, and is undefined otherwise (i.e. $i \notin [0, n - 2]$).

sequence filter operation $\text{filter}_e(s)$: $\text{filter}_e(\text{enc}(s))$, which removes every substring of $\text{enc}(s)$ between two consecutive occurrences of \dagger that does not match the regular expression e . (For instance, if $e = a^*b$ and $s = (ab, ac, aab)$, then $\text{filter}_e(\text{enc}(s)) = \dagger ab \dagger aab \dagger$.)

subsequence operation $s[i, j]$: $\text{subseq}(\text{enc}(s), i + 1, j)$, which keeps only the substring of $\text{enc}(s)$ between the $(i + 1)$ -th occurrence and the $\min(i + j, n - 1)$ -th occurrence of \dagger , if $i \in [0, n - 2]$, where n is the number of occurrences of \dagger in $\text{enc}(s)$, and is undefined otherwise (i.e. $i \notin [0, n - 2]$).

split_e(u): string operation $\text{splitstr}_e(u)$ that transforms $u = u'_1 v_1 u'_2 \cdots u'_{k-1} v_{k-1} u'_k$ into $\dagger u'_1 \dagger u'_2 \cdots \dagger u'_{k-1} \dagger u'_k \dagger$, where for each $i \in [k - 1]$, v_i is the leftmost and longest matching of e in $u'_i v_i u'_{i+1} \cdots v_{k-1} u'_k$, and u'_k does not contain any substring that matches e .

matchAll_e(u): $\text{matchAllstr}_e(u)$ that transforms $u = u'_1 v_1 u'_2 v_2 \cdots u'_{k-1} v_{k-1} u'_k$ into $\dagger v_1 \dagger \cdots \dagger v_{k-1} \dagger$, where for each $i \in [k - 1]$, v_i is the leftmost and longest occurrence of e in $u'_i v_i u'_{i+1} \cdots v_{k-1} u'_k$, moreover, u'_k does not contain any substring that matches e .

sequence read operation $\text{nth}(s, i + 1)$: $\text{elem}(\text{enc}(s), i + 1)$ that keeps only the substring between the $(i + 1)$ -th occurrence and the $(i + 2)$ -th occurrence of \dagger , if $i \in [0, n - 2]$, where n is the number of occurrences of \dagger in $\text{enc}(s)$, and is undefined otherwise (i.e. $i \notin [0, n - 2]$). (For instance, if $s = (ab, ac)$ and $i = 0$, then $\text{elem}(\text{enc}(s), 1) = \text{elem}(\dagger ab \dagger ac \dagger, 1) = ab$.)

sequence join operation $\text{join}_u(s)$: $\text{join}_u(\text{enc}(s))$ that removes the first and the last occurrences of \dagger and replaces all the remaining occurrences of \dagger by u .

sequence length operation $\text{seqlen}(s)$: $\text{seqlen}(\text{enc}(s)) - 1$ where $\text{seqlen}(\text{enc}(s))$ counts the number of occurrences of \dagger in $\text{enc}(s)$.

Hence, we obtain atomic string constraints of the following forms:

$$\begin{aligned} & \mathbf{x} = u \mid \mathbf{x} = \eta \mid \mathbf{x} = \eta \cdot \mathbf{z} \mid \mathbf{x} = \mathfrak{T}(\eta) \mid \mathbf{z} = \text{write}(\mathbf{x}, it, \eta) \mid \eta = \text{filter}_e(\mathbf{x}) \mid \\ & \eta = \text{splitstr}_e(\mathbf{x}) \mid \eta = \text{subseq}(\mathbf{x}, it_1, it_2) \mid \eta = \text{matchAllstr}_e(\mathbf{x}) \mid \eta = \text{elem}(\mathbf{x}, it) \mid \\ & \eta = \text{join}_u(\mathbf{x}) \mid it = \text{strlen}(\mathbf{x}) \mid it = \text{seqlen}(\mathbf{x}), \end{aligned}$$

where \mathfrak{T} is a finite-state transducer.

We denote by XStr the class of string constraints which are positive Boolean combinations (no negation) of the atomic string constraints, and XStr_{SL} denotes its straight-line fragment. For each formula φ in $\text{SeqStr}_{\text{SL}}$, $\text{enc}(\varphi)$ is the resulting formula in XStr . We have the following result.

Proposition 2. *For each constraint φ in SeqStr , φ and $\text{enc}(\varphi)$ are equisatisfiable. Moreover, if φ is in $\text{SeqStr}_{\text{SL}}$, then $\text{enc}(\varphi)$ is in XStr_{SL} .*

Example 2. Consider the constraint $\varphi := (\mathbf{s}_1 = \mathbf{s}_0 \cdot (u, v) \wedge \text{seqlen}(\mathbf{s}_1) < 2)$ in $\text{SeqStr}_{\text{SL}}$ where \mathbf{s}_0 is a string variable and u, v are string constants. Then, its string encoding is $\text{enc}(\varphi) := (\text{enc}(\mathbf{s}_1) = \text{enc}(\mathbf{s}_0) \cdot \dagger u \dagger v \dagger \wedge \text{seqlen}(\text{enc}(\mathbf{s}_1)) - 1 < 2)$, which is unsatisfiable, as there are at least three occurrences of \dagger in $\text{enc}(\mathbf{s}_1)$. \square

The satisfiability of $\text{SeqStr}_{\text{SL}}$ is now reduced to that of XStr_{SL} . The operations filter_e , splitstr_e , matchAllstr_e and join_u can all be represented by a finite-state transducer (see Appendix A), so the results in [19] are directly applicable. In the next section, we show that the pre-images under the remaining string operations, i.e., write , subseq , elem and seqlen , can be effectively computed.

5.2 Computing the pre-images under **write**, **subseq**, **elem** and **seqen**

We now show how to compute the pre-images under **write**, **subseq**, **elem** and **seqen**. Note that the pre-image computation utilizes an NFA \mathcal{A}_0 (resp. \mathcal{A}_1) to format the strings that represent the string sequences (resp. strings), namely, \mathcal{A}_0 (resp. \mathcal{A}_1) recognizes the language $\dagger(\Sigma\dagger)^*$ (resp. Σ^*).

Pre-image under write. Let $\mathcal{A} = (R, Q, \Sigma_\dagger, \delta, I, F)$ be a CEFA. $\text{write}^{-1}(\mathcal{L}(\mathcal{A}))$ is computed as the collection $((\mathcal{B}_{(p,q)} \cap \mathcal{A}_0, \mathcal{A}_{R_2/R}[p, q] \cap \mathcal{A}_1), t_{(p,q)})_{(p,q) \in Q \times Q}$, where $\mathcal{B}_{(p,q)}$, $\mathcal{A}_{R_2/R}[p, q]$ and $t_{(p,q)}$ are constructed as follows:

- $\mathcal{B}_{(p,q)}$ is constructed by the following procedure:
 - We create two copies of R , say R_1, R_2 . Moreover, we introduce a fresh cost register, say r' , to count the number of occurrences of \dagger .
 - We run \mathcal{A} on the input string and increase the cost register r' each time when \dagger is read. Moreover, the registers in R_1 , instead of R , are updated in the transitions.
 - When the current symbol is \dagger , we nondeterministically choose to stop increasing r' and pause the running of \mathcal{A} at the state p . Let us call this position as the pause position.
 - Finally, when reading the first \dagger after the pause position, we resume the run of \mathcal{A} at q , where the registers in R_1 (but not r') are updated.
- $\mathcal{A}_{R_2/R}[p, q]$ is obtained from $\mathcal{A}[p, q]$ by replacing each register in R with the corresponding copy in R_2 , where $\mathcal{A}[p, q]$ is the sub-automaton of \mathcal{A} that accepts the strings starting from the state p and ending at the state q .
- $t_{(p,q)} := (t_{(p,q),r})_{r \in R}$ and $t_{(p,q),r} = r^{(1)} + r^{(2)}$ where $r^{(1)}$ and $r^{(2)}$ are the copies of r for each $r \in R$.

Note that in the backward propagation of \mathcal{A} with respect to an equality $\mathfrak{z} = \text{write}(\mathfrak{x}, it, \mathfrak{y})$, the constraint $r' = it$ is added to the LIA constraint to assert that r' is equal to the index it . Formally, $\text{write}^{-1}(\mathcal{L}(\mathcal{A}))$ is a finite collection of $(\mathcal{B}_{(p,q)} \cap \mathcal{A}_0, \mathcal{A}_{R_2/R}[p, q] \cap \mathcal{A}_1, t_{(p,q)})$, where $(p, q) \in Q \times Q$, and $\mathcal{B}_{(p,q)} = (R_1 \cup \{r'\}, Q', \Sigma_\dagger, \delta', I', F')$ such that $Q' = Q \times \{pre, idle, post\}$, $I' = I \times \{pre\}$, $F' = F \times \{post\}$, and δ' comprises:

- $((q_1, pre), a, (q_2, pre), (\vec{v}, 0))$ such that $(q_1, a, q_2, \vec{v}) \in \delta$ and $a \in \Sigma$ (thus $a \neq \dagger$),
- $((q_1, pre), \dagger, (q_2, pre), (\vec{v}, 1))$ such that $(q_1, \dagger, q_2, \vec{v}) \in \delta$,
- $((q_1, pre), \dagger, (p, idle), (\vec{v}, 1))$ such that $(q_1, \dagger, p, \vec{v}) \in \delta$,
- $((p, idle), a, (p, idle), (\vec{0}, 0))$ such that $a \in \Sigma$,
- $((p, idle), \dagger, (q_2, post), (\vec{v}, 0))$ such that $(q, \dagger, q_2, \vec{v}) \in \delta$,
- $((q_1, post), a, (q_2, post), (\vec{v}, 0))$ such that $(q_1, a, q_2, \vec{v}) \in \delta$ and $a \in \Sigma$,
- $((q_1, post), \dagger, (q_2, post), (\vec{v}, 0))$ such that $(q_1, \dagger, q_2, \vec{v}) \in \delta$.

From the construction of $\mathcal{B}_{(p,q)}$, we can observe that the pair (p, q) should satisfy that in \mathcal{A} , there is a \dagger -transition into p and a \dagger -transition out of q . Moreover, q should be reachable from p according to the construction of $\mathcal{A}_{R_2/R}[p, q] \cap \mathcal{A}_1$.

Example 3. Consider the constraint $\mathbf{v} = \text{write}(\mathbf{u}, 1, \mathbf{\eta}) \wedge \mathbf{v} \in \dagger(a^+\dagger)^* \wedge \text{strlen}(\mathbf{v}) \geq 4$. Let $\mathcal{A} = (\{r_1\}, \{q_0, q_1, q_2\}, \Sigma_\dagger, \delta, \{q_0\}, \{q_1\})$ be the CEFA, where the register r_1 is used to record the string length and δ comprises the transitions $q_0 \xrightarrow[(1)]{\dagger} q_1 \xrightarrow[(1)]{a} q_2 \xrightarrow[(1)]{a} q_2 \xrightarrow[(1)]{\dagger} q_1$. Let us consider the pairs (p, q) in \mathcal{A} satisfying that there is a \dagger -transition into p and a \dagger -transition out of q , moreover, q should be reachable from p . One can easily observe that there is exactly one such pair, that is, (q_1, q_2) . Therefore, $\text{write}^{-1}(\mathcal{L}(\mathcal{A}))$ comprises exactly one tuple $(\mathcal{B}_{(q_1, q_2)} \cap \mathcal{A}_0, \mathcal{A}_{R_2/R}[q_1, q_2] \cap \mathcal{A}_1, (r_1^{(1)} + r_1^{(2)}))$ such that $\mathcal{B}_{(q_1, q_2)} = (\{r_1^{(1)}, r'\}, Q', \Sigma_\dagger, \delta', I', F')$ where

- $Q' = \{q_0, q_1, q_2\} \times \{pre, idle, post\}$, $I' = \{(q_0, pre)\}$, $F' = \{(q_1, post)\}$, and
- δ' comprises the following transitions:
 - $((q_1, pre), a, (q_2, pre), (1, 0))$, $((q_2, pre), a, (q_2, pre), (1, 0))$,
 - $((q_0, pre), \dagger, (q_1, pre), (1, 1))$, $((q_2, pre), \dagger, (q_1, pre), (1, 1))$,
 - $((q_0, pre), \dagger, (q_1, idle), (1, 1))$ (since $(q_0, \dagger, q_1, 1) \in \delta$),
 - $((q_1, idle), a', (q_1, idle), (0, 0))$ such that $a' \in \Sigma$,
 - $((q_1, idle), \dagger, (q_1, post), (1, 0))$ (since $(q_2, \dagger, q_1, 1) \in \delta$),
 - $((q_1, post), a, (q_2, post), (1, 0))$ and $((q_2, post), a, (q_2, post), (1, 0))$,
 - $((q_2, post), \dagger, (q_1, post), (1, 0))$.

To illustrate how $\mathcal{B}_{(q_1, q_2)}$ and $\mathcal{A}_{R_2/R}[q_1, q_2]$ work, we can see that $(\dagger aa \dagger a \dagger, 6)$ is accepted by \mathcal{A} , moreover, for any $b \in \Sigma$,

- $(\dagger ab \dagger a \dagger, 4, 1)$ is accepted by $\mathcal{B}_{(q_1, q_2)}$, witnessed by

$$\begin{aligned} (q_0, pre) &\xrightarrow[(1,1)]{\dagger} (q_1, idle) \xrightarrow[(0,0)]{a} (q_1, idle) \xrightarrow[(0,0)]{b} (q_1, idle) \\ &\xrightarrow[(1,0)]{\dagger} (q_1, post) \xrightarrow[(1,0)]{a} (q_2, post) \xrightarrow[(1,0)]{\dagger} (q_1, post), \end{aligned}$$

- and $(aa, 2)$ is accepted by $\mathcal{A}_{R_2/R}[q_1, q_2]$, witnessed by $q_1 \xrightarrow[(1)]{a} q_2 \xrightarrow[(1)]{a} q_2$.

Thus, \mathbf{u} and $\mathbf{\eta}$ in $\mathbf{v} = \text{write}(\mathbf{u}, 1, \mathbf{\eta})$ can be $\dagger ab \dagger a \dagger$ and aa , respectively. \square

Pre-image under subseq. Let $\mathcal{A} = (R, Q, \Sigma_\dagger, \delta, I, F)$ be a CEFA. $\text{subseq}^{-1}(\mathcal{L}(\mathcal{A}))$ is computed as $(\mathcal{B} \cap \mathcal{A}_0, t)$ such that $t := \text{true}$ and \mathcal{B} introduces two fresh registers r'_1, r'_2 to count the two numbers of occurrences of \dagger that correspond to the starting position and the length of the subsequence, respectively. Moreover, \mathcal{B} simulates the run of \mathcal{A} on the substring representing the subsequence returned by subseq . Formally, $\text{subseq}^{-1}(\mathcal{L}(\mathcal{A}))$ is computed as $(\mathcal{B} \cap \mathcal{A}_0, \text{true})$, where $\mathcal{B} = (R \cup \{r'_1, r'_2\}, Q \cup \{pre, post\}, \Sigma_\dagger, \delta', \{pre\}, \{post\})$ such that $pre, post$ are two fresh states denoting the fact that \mathcal{B} is reading a symbol before and after the subsequence respectively and δ' comprises the following transitions:

- $(pre, a, pre, (\vec{0}, 0, 0))$ such that $a \in \Sigma$ (thus $a \neq \dagger$),
- $(pre, \dagger, pre, (\vec{0}, 1, 0))$,
- $(pre, \dagger, q, (\vec{v}, 1, 0))$ such that $(p, \dagger, q, \vec{v}) \in \delta$ for some $p \in I$,
- $(q, a, q', (\vec{v}, 0, 0))$ such that $(q, a, q', \vec{v}) \in \delta$ and $a \in \Sigma$,
- $(q, \dagger, q', (\vec{v}, 0, 1))$ such that $(q, \dagger, q', \vec{v}) \in \delta$,

- $(q, \dagger, post, (\vec{v}, 0, 1))$ such that $(q, \dagger, q', \vec{v}) \in \delta$ for some $q' \in F$,
- $(post, a, post, (\vec{0}, 0, 0))$ and $(post, \dagger, post, (\vec{0}, 0, 0))$ where $a \in \Sigma$.

Note that in the backward propagation of \mathcal{A} with respect to an equality $\mathfrak{z} = \text{subseq}(\mathfrak{x}, it_1, it_2)$, the constraint $r'_1 = it_1 \wedge r'_2 = it_2$ is added to the LIA constraint to assert that the value of r'_1 and r'_2 are equal to the beginning index and the length of the subsequence respectively.

Example 4. Let $\mathfrak{v} = \text{subseq}(\mathfrak{u}, 2, 1) \wedge \mathfrak{v} \in \dagger(a^+\dagger)^* \wedge \text{strlen}(\mathfrak{v}) \geq 4$ and $\mathcal{A} = (\{r_1\}, \{q_0, q_1, q_2\}, \Sigma_\dagger, \delta, \{q_0\}, \{q_1\})$ be the CEFA, where the register r_1 is used to record the string length and δ comprises the transitions $q_0 \xrightarrow[(1)]{\dagger} q_1 \xrightarrow[(1)]{a} q_2 \xrightarrow[(1)]{a}$ $q_2 \xrightarrow[(1)]{\dagger} q_1$. Then $\text{subseq}^{-1}(\mathcal{L}(\mathcal{A}))$ is $(\mathcal{B} \cap \mathcal{A}_0, true)$ such that

$$\mathcal{B} = (\{r'_1, r'_1, r'_2\}, \{q_0, q_1, q_2, pre, post\}, \Sigma_\dagger, \delta', \{pre\}, \{post\})$$

where δ' comprises the following transitions:

- $(pre, a', pre, (0, 0, 0))$ such that $a' \in \Sigma$,
- $(pre, \dagger, pre, (0, 1, 0))$,
- $(pre, \dagger, q_1, (1, 1, 0))$ (since $(q_0, \dagger, q_1, 1) \in \delta$),
- $(q_1, a, q_2, (1, 0, 0))$, $(q_2, a, q_2, (1, 0, 0))$,
- $(q_2, \dagger, q_1, (1, 0, 1))$,
- $(q_2, \dagger, post, (1, 0, 1))$ (since $(q_2, \dagger, q_1, 1) \in \delta$ and q_1 is an accepting state in \mathcal{A}),
- $(post, a', post, (0, 0, 0))$ and $(post, \dagger, post, (0, 0, 0))$ where $a' \in \Sigma$.

To illustrate how \mathcal{B} works, we can see that $(\dagger a \dagger a a \dagger, 4, 2, 1)$ is accepted by \mathcal{B} , witnessed by

$$pre \xrightarrow[(0,1,0)]{\dagger} pre \xrightarrow[(0,0,0)]{a} pre \xrightarrow[(1,1,0)]{\dagger} q_1 \xrightarrow[(1,0,0)]{a} q_2 \xrightarrow[(1,0,0)]{a} q_2 \xrightarrow[(1,0,1)]{\dagger} post.$$

Pre-image under elem. Let $\mathcal{A} = (R, Q, \Sigma, \delta, I, F)$ be a CEFA. $\text{elem}^{-1}(\mathcal{L}(\mathcal{A}))$ is constructed similar to $\text{subseq}^{-1}(\mathcal{L}(\mathcal{A}))$. Nevertheless, the construction is slightly different since the output of elem is a string, instead of a sequence. Intuitively, $\text{elem}^{-1}(\mathcal{L}(\mathcal{A}))$ is computed as $(\mathcal{B} \cap \mathcal{A}_0, t)$ where $t := true$, a fresh register r' is introduced to count the number of occurrences of \dagger that corresponds to the position where the element is extracted. Formally, $\text{elem}^{-1}(\mathcal{L}(\mathcal{A}))$ is computed as $(\mathcal{B} \cap \mathcal{A}_0, true)$ such that $\mathcal{B} = (R \cup \{r'\}, Q \cup \{pre, post\}, \Sigma_\dagger, \delta', \{pre\}, \{post\})$, where δ' comprises the following transitions:

- $(pre, a, pre, (\vec{0}, 0))$ such that $a \in \Sigma$ (thus $a \neq \dagger$),
- $(pre, \dagger, pre, (\vec{0}, 1))$,
- $(pre, \dagger, p, (\vec{v}, 1))$ such that $p \in I$,
- $(q, a, q', (\vec{v}, 0))$ such that $(q, a, q', \vec{v}) \in \delta$ and $a \in \Sigma$ (thus $a \neq \dagger$),
- $(q, \dagger, post, (\vec{0}, 0))$ such that $q \in F$,
- $(post, a, post, (\vec{0}, 0))$ and $(post, \dagger, post, (\vec{0}, 0))$ where $a \in \Sigma$.

Note that in the backward propagation of \mathcal{A} with respect to an equality $\mathfrak{z} = \text{elem}(\mathfrak{x}, it)$, the constraint $r' = it$ is added to the LIA constraint to assert that the value of r' is equal to the index it .

Finally, we remark that the operation $\text{seqlen}(\mathfrak{x})$ can be modeled as a CEFA, similarly to the CEFA for strlen in [19].

6 Implementation and Experiments

We implement a solver $\text{OSTRICH}^{\text{SEQ}}$ on top of the string solver OSTRICH [20] and the SMT solver Princess [40], using about 5,000 lines of Scala code. The core functionality, preimage computation for dedicated string operations such as `write`, `subseq`, `elem` and `seqlen` is implemented in about 2,000 lines of code and is integrated directly into the OSTRICH framework. To support the representation of sequences, we built a lightweight library of automata which treat the separator as a special transition. The library is implemented in about 1,000 lines of code.

6.1 Benchmarks

To evaluate the effectiveness of $\text{OSTRICH}^{\text{SEQ}}$, we curate two benchmark suites, SEQBASE and SEQEXT , where SEQBASE only uses operations that are directly supported by some existing SMT solvers while SEQEXT uses some string sequence operations that are not directly supported by any existing SMT solvers. SEQBASE . The SEQBASE benchmark suite comprises SeqStr formulas that contain only the generic sequence operations, that is, sequence operations that can be applied to the elements of any type (not necessarily string), such as sequence length `seqlen`, sequence concatenation `·`, subsequence `seqt[it1, it2]`, sequence read `nth(seqt, it)` and sequence write `seqt[it → str]`. SEQBASE is designed to facilitate a fair comparison with existing solvers since they do not support the string-specific sequence operations. SEQBASE contains 140 SeqStr constraints that are generated from three templates.

- The first template $u_1 \in e_1 \wedge s_2 = s_1[n_1 \rightarrow u_1, n_2 \rightarrow u_1] \wedge u_2 = \text{nth}(s_2, m_1) \cdot \text{nth}(s_2, m_2) \wedge u_2 \in e_2 \wedge \text{seqlen}(s_2) < \text{strlen}(u_2)$ is utilized to curate 40 random instances, where u_1 is a string variable, s_1, s_2 are sequence variables, and e_1, e_2 are randomly generated regular expressions. In the first 20 instances, n_1, n_2, m_1, m_2 are integer variables while in the other 20 instances, they are random integers ranging from 0 to 10.
- The second template $u_1 \in e_1 \wedge s_2 = s_1[n_1 \rightarrow u_1, n_2 \rightarrow u_2] \wedge s_3 = s_2[n_3, len_1] \cdot s_2[n_4, len_2] \wedge u_3 = \text{nth}(s_3, m_1) \cdot \text{nth}(s_3, m_2) \cdot \text{nth}(s_3, m_3) \cdot \text{nth}(s_3, m_4) \wedge u_3 \in e_2 \wedge \text{seqlen}(s_3) < \text{strlen}(u_3) + 1$ is utilized to curate 40 random instances, where u_1, u_2 are string variables, s_1, s_2, s_3 are sequence variables, and e_1, e_2 are randomly generated regular expressions. Similar to the first template, n_i, m_i, len_1, len_2 for $i \in [1, 4]$ are integer variables in the first 20 instances but are random integers ranging from 0 to 10 in the other 20 instances.
- The third template $s_1 = s_0[m_1 \rightarrow u_1] \wedge \dots \wedge s_i = s_{i-1}[m_i \rightarrow u_i] \wedge u_1 = \text{nth}(s_1, n_1) \wedge \dots \wedge u_j = \text{nth}(s_j, n_j) \wedge u_1 \in e_1 \wedge \dots \wedge u_j \in e_j$ is utilized to curate 60 random instances, where i and j are randomly selected from the range 0 to 20, $s_0 \dots s_i$ are sequence variables, $u_1 \dots u_j$ are string variables, $e_1 \dots e_j$ are randomly generated regular expressions, and $m_1, \dots, m_i, n_1 \dots n_j$ are randomly generated integers constant ranging from 0 to 5. In this template, write ($s[\cdot \rightarrow \cdot]$) and read ($\text{nth}(s, \cdot)$) operations are performed randomly on a string sequence, and the strings read from the sequence are checked against some regular expressions.

Table 1. Experimental results on SEQBASE.

	cvc5	Z3	Princess ^{ARR}	OSTRICH ^{SEQ}
sat	48	14	52	52
unsat	87	59	77	88
solved	135	73	129	140
unknown/timeout	5	67	11	0
avg. time (s)	3.53	24.36	6.84	3.23

SEQEXT. The SEQEXT benchmark suite comprises 60 SeqStr formulas that use specific string sequence operations, namely, `filtere`, `matchAlle`, `splite` and `joinu`. Among them, 19 formulas are manually generated for the unit tests of OSTRICH^{SEQ}, and 41 formulas are generated from the real-world JavaScript programs collected from GitHub. Moreover, to compare with SMT solvers that do not support the considered string sequence operations, we rewrite them using the basic string operations whenever possible. (`matchAlle` is not rewritten since it cannot be expressed using existing string operations.)

6.2 Experimental results

We conduct experiments by comparing OSTRICH^{SEQ} with SOTA solvers, including cvc5, Z3, Z3-noodler, OSTRICH and Princess^{ARR} (referring to an array-theory-based solver implemented within Princess). SeCo [33] is excluded due to soundness issues. Additionally, `foldleft` and `map` operations are not used to simulate `join`, as most constraints return unknown under such usage in Z3. On SEQBASE, we evaluate OSTRICH^{SEQ} along with sequence solvers cvc5, Z3 and Princess^{ARR}. For SEQEXT, we rewrite the constraints using only the basic string operations and compare OSTRICH^{SEQ} with string solvers cvc5, Z3, Z3-noodler and OSTRICH.

All experiments are run on a 24×3.1 GHz-core server with 185 GB RAM, where each solver is started as a single thread, with a 60 seconds time limit and without the memory limit.

The results on SEQBASE are reported in Table 1. We can see that OSTRICH^{SEQ} outperforms all the other solvers on SEQBASE in terms of both the number of solved instances and the solving time per instance. Specifically, OSTRICH^{SEQ} solves all the 140 instances in SEQBASE, while cvc5, Z3 and Princess^{ARR} only solve 135, 73 and 129 instances, respectively. Moreover, OSTRICH^{SEQ} is more efficient than cvc5, Z3 and Princess^{ARR} (OSTRICH^{SEQ} spends 3.23 seconds in solving each instance from SEQBASE on average, while cvc5, Z3 and Princess^{ARR} spend 3.25, 24.36, and 6.84 seconds, respectively).

The results on SEQEXT are reported in Table 2. Since the SOTA sequence solvers do not support `filtere`, `matchAlle` or `splite` directly, the SeqStr constraints are rewritten to string constraints (instead of solving the SeqStr constraints directly) when we compare with them on SEQEXT. Moreover, while we can use `str.replace_re_all` to encode `filtere` and `splite`, the encoding of `matchAlle` requires

Table 2. Experimental results on SEQEXT.

		cvc5	Z3	Z3-noodler	OSTRICH	OSTRICH ^{SEQ}
SEQEXT-N	sat	19	4	4	20	23
	unsat	21	6	4	18	25
	solved	40	10	8	38	48
	unknown/timeout	8	38	40	10	0
	avg. time (s)	10.83	0.13	0.05	15.92	3.53
SEQEXT-M	sat	-	-	-	-	12
	unsat	-	-	-	-	0
	solved	-	-	-	-	0
	unknown/timeout	-	-	-	-	0
	avg. time (s)	-	-	-	-	1.77

finite-state transducers which have not been supported by most SOTA string solvers. As a result, we do *not* generate the string constraints for the 12 instances in SEQEXT that contain `matchAlle`, which are denoted by SEQEXT-M, whereas the rest 48 instances are denoted by SEQEXT-N.

From Table 2, we can see that Z3 and Z3-noodler solve only 10 and 8 out of 48 constraints in SEQEXT-N, respectively, and cvc5 and OSTRICH solve 40 and 38 instances, respectively, indicating that the complexity of the string constraints generated from the `SeqStr` constraints. On the other hand, OSTRICH^{SEQ} solves all of them, demonstrating its superiority in solving the `SeqStr` constraints with string sequence operations. For efficiency, OSTRICH^{SEQ} spends 3.53 seconds per instance on average, significantly less than cvc5 and OSTRICH (10.83 and 15.92 seconds on average, respectively). For the constraints in SEQEXT-M, OSTRICH^{SEQ} successfully solves all 12 instances with an average time of 1.77 seconds. As aforementioned, for the constraints in SEQEXT-M, we do not generate string constraints or compare against cvc5, Z3, Z3-noodler and OSTRICH, as they cannot be expressed using only the basic operations of string theory.

7 Related work

String constraint solving. String constraint solving has received considerable attention in the literature. Amadini [5] provides a comprehensive survey of the literature up to 2021. In particular, prior approaches are classified into three main categories: automata-based (relying on finite automata to represent the domain of string variables and to handle string operations, e.g., [3,31,17,28,37,12,22]), word-based (algebraic approaches based on systems of word equations, e.g., [36,7,25,51,11,39,47,46,12,22]), and unfolding-based approaches (explicitly reducing each string into a number of contiguous elements denoting its characters, e.g., [43,34,42,35,6,24]).

From another perspective, there are two lines of work aiming for building practical string constraint solvers. (1) One could support as many operations as possible, but primarily resort to heuristics, offering no completeness/termination

guarantees. Many string constraint solvers are implemented in SMT solvers, allowing combination with other theories, most commonly the theory of integers for string lengths. Some (non-exhaustive) examples include `cvc4/5` [36,7], `Z3` [25,13], `Z3-str2/3/4` [51,11,39], `S3(P)` [47,46], `Trau` [2] (or its variants `Trau+` [4]), `Slent` [48], etc. More recent ones include `Z3str3RE` [12] and `Z3-Noodler` [22] (which is based on stabilization-based algorithm [15,21] for solving word equations with regular constraints). (2) The second approach is to develop solvers for decidable fragments, including `Norn` [3], `SLOTH` [31] and `OSTRICH` [17,20], which are usually based on complete decision procedures (e.g. [28,4,37])

We also mention that there are solvers which emphasize certain aspects of string constraint solving by providing dedicated methods or optimization. These include, for instance, those for more expressive regular expressions [10,18], regex-counting [32], string/integer conversion and flat regular constraints [49], Not-Substring Constraint [1], integer data type [19], etc.

Sequences as an extension of strings. Strings and sequences are closely related, but may exhibit in different forms. For instance, a string can be considered as a sequence over a finite alphabet. Jez et al. [33] recently studied sequence theories which are an extension of theories of strings with an infinite alphabet of letters, together with a corresponding alphabet theory (e.g. linear integer arithmetic). They provided decision procedures based on parametric automata/transducers, giving rise to a sequence constraint solver `SeCo`. Our work is different in that we essentially work on a sequence theory where the element is instantiated by the string type.

Theory of arrays. There are many studies on solving formulas in the theory of arrays or its extensions, for instance, [45,16,30,26,23,27], which are also related to the sequence theory. Note that in these studies, the elements are considered to be either of a generic type or of integer type. In other words, the theory of arrays where the elements are of the string type has not been defined and investigated therein.

8 Conclusion

In this paper, we have proposed `SeqStr`, a logic of string sequences, which supports a wealth of string and sequence operations—especially their interactions—commonly in real-world string-manipulation programs. We have provided a decision procedure for the straight-line fragment, which is implemented as a new tool `OSTRICHSEQ`. The experiments on both hand-crafted and real-world benchmarks demonstrate that `OSTRICHSEQ` is an effective and promising tool for testing, analysis and verification of string-manipulation programs in practice.

A Appendix: Representing filter_e , splitstr_e , matchAllstr_e and join_u by NFT

The semantics of regular expression matching, like first match and longest match, can be represented by transducers (e.g., refer to [38]). While any matching semantics could be used, we specifically illustrate the construction of transducers utilizing left and longest match semantics. The construction detailed in this section also utilizes ϵ -transitions in (nondeterministic) finite transducers (NFTs), which does not exceed the expressive capabilities of the NFTs described in Section 2.

A.1 Representing filter_e by NFT

Recall that $\text{filter}_e(s)$ removes every substring of s between two consecutive occurrences of \dagger that does not match the regular expression e . To build the NFT that represents filter_e , we assume that the automata $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ and $\bar{\mathcal{A}} = (\bar{Q}, \Sigma, \bar{\delta}, \bar{I}, \bar{F})$ are already built to identify the language of e and \bar{e} , respectively. We then introduce four states: q_n for the start point of searching on s , q_b for beginning to match e , q_f for a failed match of e , and q_t for the terminal of the searching. The core idea is to nondeterministically select either e or \bar{e} for a substring between two \dagger symbols and output the substring if it matches e successfully; otherwise output an empty string. Formally, the NFT illustrating filter_e is defined as $(\{q_n, q_b, q_f, q_t\} \cup Q \cup \bar{Q}, \Sigma_{\dagger}, \delta', \{q_n\}, \{q_t\})$ where δ comprises tuples that correspond to the matching of e

- $(q_n, \dagger, q_b, \dagger)$,
- $(q_b, \epsilon, q_I, \epsilon)$ such that $q_I \in I$,
- (q, a, q', a) such that $(q, a, q') \in \delta$,
- $(q_F, \dagger, q_b, \dagger)$ such that $q_F \in F$,
- $(\bar{q}_F, \dagger, q_b, \dagger)$ such that $\bar{q}_F \in \bar{F}$,

and tuples that indicate a mismatch of e

- $(q_n, \dagger, q_f, \epsilon)$,
- $(q_f, \epsilon, \bar{q}_I, \epsilon)$ such that $\bar{q}_I \in \bar{I}$,
- $(\bar{q}, a, \bar{q}', \epsilon)$ such that $(\bar{q}, a, \bar{q}') \in \bar{\delta}$,
- $(q_F, \dagger, q_f, \epsilon)$ such that $q_F \in F$,
- $(\bar{q}_F, \dagger, q_f, \epsilon)$ such that $\bar{q}_F \in \bar{F}$,

and tuples that terminate the search on s

- $(q_n, \dagger, q_t, \dagger)$,
- $(q_F, \dagger, q_t, \dagger)$ such that $q_F \in F$,
- $(\bar{q}_F, \dagger, q_t, \dagger)$ such that $\bar{q}_F \in \bar{F}$.

A.2 Representing matchAllstr_e by NFT

Recall that $\text{matchAllstr}_e(u)$ transforms $u = u'_1 v_1 u'_2 v_2 \cdots u'_{k-1} v_{k-1} u'_k$ into $\dagger v_1 \dagger \cdots \dagger v_{k-1} \dagger$, where for each $i \in [k-1]$, v_i is the leftmost and longest occurrence of e in $u'_i v_i u'_{i+1} \cdots v_{k-1} u'_k$, moreover, u'_k does not contain any substring that matches e . Here, we must preserve the leftmost and longest match semantics. Therefore, we employ three modes to show the matching situations: **NotMatch** to indicate no match for e , **Matching** to signify that e is currently matched, and **EndMatch** to denote the completion of matching e . Assume that $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ is the automaton that recognizes e , $Q' = \{\langle \text{mode}, \text{cur}, \text{noreach} \rangle \mid \text{mode} \in \{\text{NotMatch}, \text{Matching}, \text{EndMatch}\}, \text{cur} \in 2^Q, \text{noreach} \in 2^Q\}$ is the primary states set in the NFT representing matchAllstr_e . The intuition is utilizing cur to denote the states matched currently and noreach to denote the states unable to reach final states, thereby ensuring

- whenever the NFT transitions from mode **NotMatch** to **Matching**, any current states must not reach final states (otherwise we do not match leftmost),
- whenever the NFT transitions from mode **Matching** to **EndMatch**, any current states must not reach final states again (otherwise we do not match longest).

Let $\text{next}_a(Q_0) = \{q' \mid (q, a, q') \in \delta, q \in Q_0\}$ denote the set of states reachable from Q_0 after reading label a within \mathcal{A} , then the NFT illustrating matchAllstr_e is $(Q' \cup \{q_f\}, \Sigma_\dagger, \delta', \langle \text{NotMatch}, \emptyset, \emptyset \rangle, \{q_f\})$ where F' comprises states δ' comprises tuples that indicate the termination of the search

- $(\langle \text{NotMatch}, \emptyset, \text{noreach} \rangle, \epsilon, q_f, \dagger)$ such that $\text{noreach} \cap F = \emptyset$,
- $(\langle \text{EndMatch}, \text{cur}, \text{noreach} \rangle, \epsilon, q_f, \dagger)$ such that $\text{noreach} \cap F = \emptyset$ and $\text{cur} \in 2^Q$,

and tuples illustrating the transformation among these three modes

- $(\langle \text{NotMatch}, \emptyset, \text{noreach} \rangle, a, \langle \text{NotMatch}, \emptyset, \text{next}_a(\text{noreach} \cup I) \rangle, \epsilon)$,
- $(\langle \text{NotMatch}, \emptyset, \text{noreach} \rangle, a, \langle \text{Matching}, \text{next}_a(I), \text{next}_a(\text{noreach}) \rangle, \dagger a)$ such that $\text{next}_a(I) \neq \emptyset$,
- $(\langle \text{NotMatch}, \emptyset, \text{noreach} \rangle, a, \langle \text{EndMatch}, \text{next}_a(I), \text{next}_a(\text{noreach}) \rangle, \dagger a)$ such that $\text{next}_a(I) \cap F \neq \emptyset$,
- $(\langle \text{Matching}, \text{cur}, \text{noreach} \rangle, a, \langle \text{Matching}, \text{next}_a(\text{cur}), \text{next}_a(\text{noreach}) \rangle, a)$ such that $\text{next}_a(\text{cur}) \neq \emptyset$,
- $(\langle \text{Matching}, \text{cur}, \text{noreach} \rangle, a, \langle \text{EndMatch}, \text{next}_a(\text{cur}), \text{next}_a(\text{noreach}) \rangle, a)$ such that $\text{next}_a(\text{cur}) \cap F \neq \emptyset$,
- $(\langle \text{EndMatch}, \text{cur}, \text{noreach} \rangle, a, \langle \text{NotMatch}, \emptyset, \text{next}_a(\text{cur} \cup \text{noreach} \cup I) \rangle, \epsilon)$,
- $(\langle \text{EndMatch}, \text{cur}, \text{noreach} \rangle, a, \langle \text{Matching}, \text{next}_a(I), \text{next}_a(\text{cur} \cup \text{noreach}) \rangle, \dagger a)$ such that $\text{next}_a(I) \neq \emptyset$,
- $(\langle \text{EndMatch}, \text{cur}, \text{noreach} \rangle, a, \langle \text{EndMatch}, \text{next}_a(I), \text{next}_a(\text{cur} \cup \text{noreach}) \rangle, \dagger a)$ such that $\text{next}_a(I) \cap F \neq \emptyset$.

A.3 Representing splitstr_e and join_u by NFT

$\text{splitstr}_e(u)$ transforms $u = u'_1 v_1 u'_2 \cdots u'_{k-1} v_{k-1} u'_k$ into $\dagger u'_1 \dagger u'_2 \cdots u'_{k-1} \dagger u'_k \dagger$, where for each $i \in [k-1]$, v_i is the leftmost and longest matching of e in $u'_i v_i u'_{i+1} \cdots v_{k-1} u'_k$.

$\text{splitstr}_e(u)$ can be simulated by $\dagger \cdot \text{replaceAll}_e(u, \dagger) \cdot \dagger$. Here, $\text{replaceAll}_e(u, \dagger)$ denotes the function replacing each occurrence of e in u with \dagger , which can be seen as a transducer [17].

$\text{join}_u(v)$ removes the first and the last occurrences of \dagger and replaces all the remaining occurrences of \dagger in v by u . Similar to splitstr_e , $\text{join}_u(v)$ can be emulated using replaceAll function along with concatenation, e.g., $v' = \text{join}_u(v)$ can be transformed to $v = \dagger \cdot v_{tmp} \cdot \dagger \wedge v' = \text{replaceAll}_{\dagger}(v_{tmp}, u)$.

Considering the observation mentioned above, the NFTs that represent splitstr_e and join_u are straightforward.

References

1. P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, D. Hu, W. Tsai, Z. Wu, and D. Yen. Solving not-substring constraint with flat abstraction. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 305–320, 2021.
2. P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer. Flatten and conquer: a framework for efficient analysis of string constraints. In *PLDI*, pages 602–617, 2017.
3. P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *CAV*, pages 150–166, 2014.
4. P. A. Abdulla, M. F. Atig, B. P. Diep, L. Holík, and P. Janku. Chain-free string constraints. In *ATVA*, pages 277–293, 2019.
5. R. Amadini. A survey on string constraint solving. *ACM Comput. Surv.*, 55(2):16:1–16:38, 2023.
6. R. Amadini, G. Gange, and P. J. Stuckey. Propagating lex, find and replace with dashed strings. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 18–34, Cham, 2018. Springer International Publishing.
7. H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 415–442, 2022.
8. C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.7. Technical report, Department of Computer Science, The University of Iowa, 2025. Available at www.SMT-LIB.org.
9. C. Barrett and C. Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.
10. M. Berzish, J. D. Day, V. Ganesh, M. Kulczynski, F. Manea, F. Mora, and D. Nowotka. Towards more efficient methods for solving regular-expression heavy string constraints. *Theor. Comput. Sci.*, 943:50–72, 2023.
11. M. Berzish, V. Ganesh, and Y. Zheng. Z3str3: A string solver with theory-aware heuristics. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 55–59, 2017.
12. M. Berzish, M. Kulczynski, F. Mora, F. Manea, J. D. Day, D. Nowotka, and V. Ganesh. An SMT solver for regular expressions and linear arithmetic over string length. In *Computer Aided Verification (CAV)*, pages 289–312, 2021.
13. N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, pages 307–321, 2009.

14. N. Bjørner, V. Ganesh, R. Michel, and M. Veanes. An SMT-LIB format for sequences and regular expressions. In *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories (SMT)*, 2012.
15. F. Blahoudek, Y. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síc. Word equations in synergy with regular constraints. In *Formal Methods (FM)*, pages 403–423, 2023.
16. A. R. Bradley, Z. Manna, and H. B. Sipma. What’s decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 427–442, 2006.
17. T. Chen, Y. Chen, M. Hague, A. W. Lin, and Z. Wu. What is decidable about string constraints with the replaceall function. *PACMPL*, 2(POPL):3:1–3:29, 2018.
18. T. Chen, A. Flores-Lamas, M. Hague, Z. Han, D. Hu, S. Kan, A. W. Lin, P. Rümmer, and Z. Wu. Solving string constraints with regex-dependent functions through transducers with priorities and variables. *Proc. ACM Program. Lang.*, 6(POPL):1–31, 2022.
19. T. Chen, M. Hague, J. He, D. Hu, A. W. Lin, P. Rümmer, and Z. Wu. A decision procedure for path feasibility of string manipulating programs with integer data type. In *Automated Technology for Verification and Analysis (ATVA)*, pages 325–342, 2020.
20. T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.*, 3(POPL), 2019.
21. Y. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síc. Solving string constraints with lengths by stabilization. *Proc. ACM Program. Lang.*, 7(OOPSLA2):2112–2141, 2023.
22. Y. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síc. Z3-noodler: An automata-based string solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 24–33, 2024.
23. P. Daga, T. A. Henzinger, and A. Kupriyanov. Array folds logic. In *Computer Aided Verification*, pages 230–248, Cham, 2016.
24. J. D. Day, T. Ehlers, M. Kulczynski, F. Manea, D. Nowotka, and D. B. Poulsen. On solving word equations using sat. In *Reachability Problems*, pages 93–106, Cham, 2019. Springer International Publishing.
25. L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.
26. S. Falke, F. Merz, and C. Sinz. Extending the theory of arrays: memset, memcpy, and beyond. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 108–128, 2013.
27. B. Farinier, R. David, S. Bardin, and M. Lemerre. Arrays made simpler: An efficient, scalable and thorough preprocessing. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 57 of *EPiC Series in Computing*, pages 363–380, 2018.
28. V. Ganesh, M. Minnes, A. Solar-Lezama, and M. C. Rinard. Word equations with length constraints: What’s decidable? In *HVC*, pages 209–226, 2012.
29. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Dpll(t): Fast decision procedures. In *Computer Aided Verification*, pages 175–188, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
30. P. Habermehl, R. Iosif, and T. Vojnar. What else is decidable about integer arrays? In *Foundations of Software Science and Computational Structures (FOSSACS)*, pages 474–489, 2008.

31. L. Holík, P. Janku, A. W. Lin, P. Rümmer, and T. Vojnar. String constraints with concatenation and transducers solved efficiently. *PACMPL*, 2(POPL):4:1–4:32, 2018.
32. D. Hu and Z. Wu. String constraints with regex-counting and string-length solved more efficiently. In *Dependable Software Engineering. Theories, Tools, and Applications (SETTA)*, pages 1–20, 2023.
33. A. Jez, A. W. Lin, O. Markgraf, and P. Rümmer. Decision procedures for sequence theories. In *Computer Aided Verification (CAV)*, pages 18–40, 2023.
34. A. Kiezun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4), Feb. 2013.
35. G. Li and I. Ghosh. Pass: String solving with parameterized array and interval automaton. In *Hardware and Software: Verification and Testing*, pages 15–31, Cham, 2013. Springer International Publishing.
36. T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV*, pages 646–662, 2014.
37. A. W. Lin and P. Barceló. String solving with word equations and transducers: Towards a logic for analysing mutation XSS. In *Principles of Programming Languages (POPL)*, pages 123–136, 2016.
38. Y. Minamide, Y. Sakuma, and A. Voronkov. Translating regular expression matching into transducers. In *2010 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 107–115, 2010.
39. F. Mora, M. Berzish, M. Kulczynski, D. Nowotka, and V. Ganesh. Z3str4: A multi-armed string solver. In *Formal Methods (FM)*, pages 389–406, 2021.
40. P. Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, LNCS, pages 274–289, 2008.
41. P. Rümmer. Princess - the scala theorem prover. <https://philipp.ruemmer.org/princess.shtml>, 2024. Latest release: 2024-11-08.
42. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Security and Privacy (SP)*, page 513–528, USA, 2010. IEEE Computer Society.
43. J. D. Scott, P. Flener, J. Pearson, and C. Schulte. Design and implementation of bounded-length sequence variables. In *Integration of AI and OR Techniques in Constraint Programming*, pages 51–67, Cham, 2017. Springer International Publishing.
44. Y. Sheng, A. Nötzli, A. Reynolds, Y. Zohar, D. Dill, W. Grieskamp, J. Park, S. Qadeer, C. Barrett, and C. Tinelli. Reasoning about vectors using an smt theory of sequences. In *Automated Reasoning*, pages 125–143, Cham, 2022. Springer International Publishing.
45. A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *Logic in Computer Science (LICS)*, pages 29–37. IEEE Computer Society, 2001.
46. M. Trinh, D. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *CCS*, pages 1232–1243, 2014.
47. M. Trinh, D. Chu, and J. Jaffar. Progressive reasoning over recursively-defined strings. In *Computer Aided Verification (CAV)*, pages 218–240, 2016.
48. H.-E. Wang, S.-Y. Chen, F. Yu, and J.-H. R. Jiang. A symbolic model checking approach to the analysis of string and length constraints. In *ASE*, page 623–633. ACM, 2018.

- 49. H. Wu, Y. Chen, Z. Wu, B. Xia, and N. Zhan. A decision procedure for string constraints with string/integer conversion and flat regular constraints. *Acta Informatica*, 61(1):23–52, 2024.
- 50. Z3Prover. Sequences | online z3 guide, 2025.
- 51. Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods Syst. Des.*, 50(2-3):249–288, 2017.